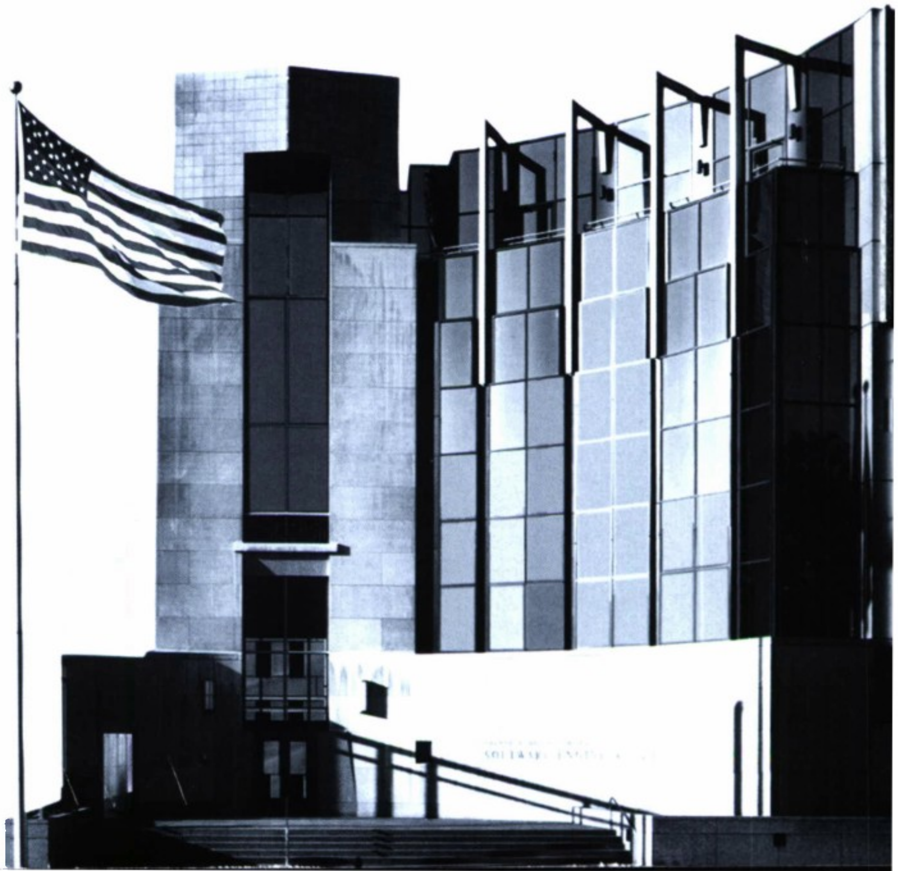
Carnegie Mellon
Software Engineering Institute

Software Architecture Documentation in Practice: Documenting Architectural Layers

Felix Bachmann
Len Bass
Jeromy Carriere
Paul Clements
David Garlan
James Ivers
Robert Nord
Reed Little

March 2000

SPECIAL REPORT
CMU/SEI-2000-SR-004



Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Software Architecture Documentation in Practice: Documenting Architectural Layers

CMU/SEI-2000-SR-004

Felix Bachmann
Len Bass
Jeromy Carriere
Paul Clements
David Garlan
James Ivers
Robert Nord
Reed Little

March 2000

Architecture Tradeoff Analysis Initiative

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Norton L Compton

Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

Abstract

This report represents the first milestone of a work in progress. That work is a comprehensive handbook on how to produce high-quality documentation for software architectures. The handbook, tentatively entitled *Software Architecture Documentation in Practice*, will be published in mid- to late-2000 by Addison Wesley Longman as a book in the SEI series on software engineering. Aimed squarely at the practitioner, the handbook is intended to fill a gap in the literature: There is a complete lack of language-independent guidance about how to actually capture an architecture in written form so that it can fulfill its purpose as a communication vehicle providing a unified design vision to all of the varied stakeholders of a development project.

The theme of the work is that documenting an architecture entails documenting the set of relevant views of that architecture, and then completing the picture with documentation of information that transcends any single view. The report lays out our approach and organization for the complete book, and provides full guidance for one of the most commonly used architectural views: the layer diagram. The audience for this book is the community of practicing architects, apprentice architects, and developers who are on the receiving end of architectural documentation.

Preface to the Special Report

This special report represents the first milestone of a work in progress. That work is a comprehensive handbook on how to produce high-quality documentation for software architectures. The handbook, tentatively entitled *Software Architecture Documentation in Practice,* will be published in mid- to late-2000 by Addison Wesley Longman as a book in the Software Engineering Institute (SEI) series on software engineering. Aimed squarely at the practitioner, the handbook is intended to fill a gap in the literature. There is no shortage of material on the importance of architecture. There is less, but still plentiful, material on tools for crafting an architecture well-suited to its purpose through the use of styles and patterns. And there is an over-abundance of material available on how to use particular design notations such as the Unified Modeling Language (UML) to specify designs. But there is a complete lack of language-independent guidance about how to actually capture an architecture in written form so that it can fulfill its purpose as a communication vehicle providing a unified design vision to all of the varied stakeholders of a development project.

The theme of the work is that documenting an architecture entails documenting the set of relevant views of that architecture, and then completing the picture with documentation of information that transcends any single view. What are the relevant views? It depends on your goals. Architecture documentation can serve many purposes: a mission statement for implementors, a basis for analysis, the specification for automatic code generation, the starting point for system understanding and asset recovery, or the blueprint for project planning. Different views support different goals and uses, and so another tenet of documentation is that what you write down depends on what you expect to do with the architecture.

We envision a book in the 300-page range that conveys the following information:

- Uses of software architecture documentation. How one documents depends on how one wishes to use the documentation. We will lay out the possible end goals for architecture documentation, and provide documentation strategies for each.
- Architectural views. We view documenting software architecture primarily as documenting the relevant views, and then augmenting this information with relevant trans-views information. The heart of the book will be an introduction to the two dozen or so most relevant architectural views (grouped into major families) along with practical guidance about how to write them down. Examples will be included for each.
- Documenting architectural styles. Styles and patterns have emerged as important tools in the architect's repertoire, and since many styles and patterns transcend single structures

(and often do so either unintentionally or ambiguously) we include a section on how to document architectural styles and patterns.

- Validating documentation. Once documentation has been created, it should be validated before being turned over to those stakeholders who depend on its quality. We will give a practical method for reviewing and validating architectural documentation.

We will organize the information so that the reader can quickly get the information needed to accomplish the task at hand. In particular, we will ask the reader to explicitly choose the usage planned for the software architecture documentation. Then we will direct him/her to the particular structures and styles information that best serves that usage.

The audience for this book is the community of practicing architects, apprentice architects, and developers who are on the receiving end of architectural documentation.

The special report lays out our approach and organization for the complete book, and provides full guidance for one of the most commonly used architectural views: the *layer diagram*. The primary purpose of this document is to serve as review fodder for the full handbook. Therefore, the material that follows this preface is written exactly as though it were in the book itself—you'll notice references to "this book" and the like sprinkled throughout the text. At places like this, we ask you to "play along" and pretend you're reading the final work. We earnestly solicit your opinions about what you see. You can provide feedback by sending email with your comments to clements@sei.cmu.edu.

Preface to *Software Architecture Documentation in Practice*

What This Book Is About

Software architecture is enjoying a flurry of attention these days. A new book about it seems to pop out monthly. In response to industrial need, universities are adding software architecture to their software engineering curricula. It's not unusual for "software architect" to be a defined position in organizations, and professional practice groups for software architects are emerging. It has been the subject of international conferences and workshops. The purveyors of the Unified Modeling Language (UML) promote their product by calling it "the standard notation for software architecture," a claim we think says at least as much about the pervasiveness of architecture as the language. At the Software Engineering Institute (SEI) we maintain a bibliography of papers and books about software architecture; its population is closing in on 1000.

You'd think that in all this blizzard of information, someone by now would have figured out how to write down a software architecture and shared that information with the rest of us.

Wouldn't you?

Surprisingly, little practical guidance is available for how to capture an architecture that is independent of a particular language or notation. To be sure, a pile of books exist about how to use a particular language—again, UML comes to mind—but it seemed to us that these are all written as though the language were the important thing and that what you wanted to represent in the language was somehow secondary. We view any language or notation as but a means to an end, and we wanted to provide guidance that viewed the architecture as the first-class citizen, with language relegated to its more appropriate role as the documentation endeavor's junior partner.

So we decided that there might be room for a modest little book that helped you decide what information about an architecture was important to capture, and then suggested notations and gave examples for capturing it.

First, let's agree on some basic context. There are many definitions of software architecture, but we like this one: A software architecture for a system is the structure or structures of the system, which comprise components, their externally visible behavior, and the relationships among them [Bass 98].

Let's also agree, without much elaboration, that a software architecture, especially for large systems, is a critical success factor. You cannot hope to build an acceptable system unless the architecture is appropriate, suitable, and effectively communicated. If you don't believe that, you're in the wrong place; we suggest you peruse some of the "For Further Reading" suggestions throughout the book and come back when you've been converted.

But just to summarize:

- The architecture serves as the blueprint for both the system and the project developing it. It defines the work assignments that must be carried out by separate design and implementation teams.
- The architecture is the carrier of system qualities such as performance, modifiability, and security, none of which can be achieved without a unifying architectural vision.
- Architecture is a vehicle for early analysis to make sure that the design approach will yield an acceptable system.
- And architecture is the artifact that holds the key to post-deployment system understanding or mining efforts.

In short, architecture is the conceptual glue that holds every phase of the project together.

An obvious truth is that the most perfect architecture is useless if it is not understood or (perhaps worse) misunderstood. Documenting the architecture is the critical, crowning step to crafting the architecture.

We intend this book to be a practitioner-oriented guide to the different kinds of information that constitutes an architecture. We wanted to give practical guidance for choosing which information should be documented, and show (with examples in various notations, including UML) how to describe that information in writing so that others may use it to carry out their architecture-based work: implementation, analysis, recovery, etc. Therefore, we cover the following:

- Uses of software architecture documentation. How one documents depends on how one wishes to use the documentation. We lay out possible end goals for architecture documentation, and provide documentation strategies for each.
- Architectural views. We hold that documenting software architecture is primarily about documenting the relevant views, and then augmenting this information with relevant trans-view information. The heart of the book is an introduction to the most relevant architectural views (grouped into a small number of major families) along with practical guidance about how to write them down. Examples are included for each.

- Validating documentation. Once documentation has been created, it should be validated before being turned over to those stakeholders who depend on its quality. We give a practical method for reviewing and validating architectural documentation.

The audience for this book is the community of practicing architects, apprentice architects, and developers who are on the receiving end of architectural documentation.

Reader's Guide

The material is presented in the following parts.

Documenting Software Architectures. This chapter explains what software architecture is, what uses it has, why it needs to be written down to realize its full potential, and good qualities of architecture documentation.

The View Zoo. Architectural views form a splendid basis for architecture documentation. Unfortunately, there are over two dozen views that occur in the literature, many of which overlap and are confusingly explained and many of which are only relevant in particular circumstances. We've tried to bring some order to the chaos by organizing the collection into a manageable number of groups that we call "the view zoo." The view zoo introduces the commonly used views and describes each using a pattern template. We explain what each view is well suited (and not well suited) for; for example, the process view is good for reasoning about a system's performance. We introduce or define notations for each view, and discuss variations of the view that have appeared in the literature—often the same or very similar views are prescribed by different authors using different names. For each view, we show an example. We generally, but not always, use UML to illustrate the views.

This chapter also relates views to each other. Although the possible combinations are endless, there are a manageable number of specific view combinations that are of use to practitioners. Often, more than one view can be profitably combined to show information that is a function of the combination.

Finally, this chapter complements "the view zoo" with "the trace space." Traces embody sequential paths through the architectural structures illuminated by the views. Scenarios are often used to describe traces, as are message sequence charts. Traces allow a view to "come alive"; they serve as excellent tools for understanding how the design will behave when the system is running.

Documentation Beyond Views. Although capturing architectural views is the essence of architectural documentation, a complete treatment requires writing down information that applies to the collection of views as a whole. Chief among that information is rationale, explaining why the architect made the decisions that combined to form the architecture. Rationale applies to individual views, but there is also the rationale that explains how the overall architecture is in fact a solution to its requirements. Other holistic information includes system context, architectural constraints, and architectural quality goals.

Practitioner's Guide to Documenting Architecture. Given the intended usage of an architecture (analysis, reconstruction, achieving common understanding, basis for deriving code, etc.), how should you go about documenting it? What views and styles should you consider documenting? What other information should you capture?

Validating Architecture Documentation. As the pre-eminent design artifact, architectural documentation carries the burden of project success. As such, it should be subject to validation (review). This chapter introduces the principle of active design reviews as the best-of-breed review methods, and gives example review forms for the architectural views and styles we have presented earlier.

Final Word

We believe strongly in the importance of architecture in building successful systems. Architecture serves as a communication vehicle for negotiating trade-offs, handing off detailed design and implementation tasks, performing early analysis, tracking progress, and conveying system understanding. But no architecture can achieve any of these goals if it is not effectively communicated, and documentation is the key to successful communication. We hope we have provided a useful handbook for practitioners in the field.

1 Documenting Software Architectures

Software architecture has emerged as an important sub-discipline of software engineering, particularly in the realm of large system development. While there is no universal definition of software architecture, there is no shortage of them, either. The following are a few of the most-cited ones:

- Bass, Clements, and Kazman, 1998: *The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. By "externally visible" properties, we are referring to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on* [Bass 98].
- Garlan and Perry, 1995: *The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time* [Garlan 95].
- Garlan and Shaw, 1993: *...beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives* [Garlan 93].
- Perry and Wolf, 1992: *A set of... design elements that have a particular form* [Perry 92].

What these definitions have in common is their emphasis on architecture as a description of a system as a sum of smaller parts, and how those parts relate to and cooperate with each other to perform the work of the system. Architecture gives us intellectual control over the very complex by allowing us to substitute the complex with a set of interacting pieces, each one of which is substantially simpler than the whole.

The prudent partitioning of a whole into parts is what allows groups of people—often groups of groups of people separated by organizational, geographical, and even temporal boundaries—to work cooperatively and productively together to solve a much larger problem than any of them would be capable of individually. It's "divide and conquer" followed by "mind your own business"—that is, each part can be built knowing very little about the other parts—resulting in "e pluribus unum."

No less important is the fact that the quality attributes desired for a system are largely carried by the architecture. Do you require high performance? Then you need to be concerned with the decomposition of the work into cooperating processes and you need to manage the inter-process communication volume and data access frequencies. Does your system need high accuracy? Then you must worry about the inter-component data flow. Security? Then you need to legislate usage relationships and communication restrictions among the components, and you may need to introduce special, trusted components. Modifiability and portability? Then prudent separation of concerns among the components is paramount. Do you want to be able to field the system incrementally, by releasing successively larger subsets? Then you have to keep the dependency relationships among the pieces untangled in order to avoid the "nothing works until everything works" syndrome. All of these concerns and their solutions are purely architectural in nature.

The full treatment of software architecture—how to build one, how to evaluate one to make sure it's a good one, how to recover one from a jumble of legacy code, and how to drive a development effort once you have one—is covered elsewhere. An armful of books exist on the topic of architecture, and more are appearing on a regular basis. Several of them are mentioned in the "For Further Reading" sections throughout the book.

But these topics are not the topic of this book. This book is aimed at a small but critical niche in the big picture of software architecture that is this: *How do you write down an architecture so that others can use it and maintain it?* Given the uses of architecture, this is quite an important question. If architecture is the blueprint that allows teams to work together, the project will fall apart (by failing to meet its schedules, behavioral requirements, or quality goals) if the blueprint is so poor that no one can read it. If you go to the trouble of creating a robust architecture, you *must* go to the trouble of describing it in enough detail, without unintentional ambiguity, and in an organization so that others can quickly find needed information. Otherwise your effort will have been wasted, because the architecture will be unusable.

1.1 Rules for Sound Documentation

Architecture documentation is in many ways akin to the documentation we write in other facets of our software development projects. As such, it obeys the same fundamental rules for what sets apart good, usable documentation from poor, ignored documentation.

1. Documentation should be written from the point of view of the reader, not the writer.

This rule seems so obvious, but it is surprising how seldom it seems to be considered. First of all, it is a matter of arithmetic: A document is written approximately once (a little more than that if you count the time for revisions). We hope it is read many scores of times. Therefore, the document's "efficiency" is optimized if we make things easier for the reader. Edsger Dijkstra, the inventor of many of the software engineering principles we now take for granted, once said that he will happily spend two hours pondering how to make a single sentence clearer. He reasons that if the paper is read by a couple of hundred people—a decidedly modest estimate for someone of Dijkstra's caliber—and he can save each reader a minute or two of confusion, then it's well worth the effort. Professor Dijkstra's consideration for the reader reflects his classic old-world manners, which brings us to the second argument: Writing for the reader is just plain polite. A reader who feels like the document was written with him or her in mind will appreciate the effort, but more to the point, will come back to the document again the next time they need information about its subject. Which brings us to the third argument: Documents written for the reader will be read; documents written for the convenience of the writer will not be. It's the same reason we like to shop at stores that seem to want our business, and avoid stores that do not.

In the realm of software documentation, documents written for the writer often take one of two forms: Stream of consciousness and stream of execution. Stream of consciousness writing captures thoughts in the order in which they occurred to the writer. Stream of consciousness writing can be avoided by making sure that you know what question(s) are being answered by each section of a document. Stream of execution writing captures thoughts in the order in which they occur during the execution of a software program. For certain kinds of software documentation, this is entirely appropriate, but it should never be given as the whole story.

Corollaries include the following:

- Documentation should be organized for ease of reference, not ease of reading. A document may be read from cover to cover at most once, and probably never. But a document is likely to be referenced hundreds or thousands of times. Hence, the same three arguments above apply again.
- Mark what you don't yet know with "to be determined" rather than leaving it blank. Many times we can't fill in a document completely because we don't yet know the information or because decisions affecting it have not been made. In that case, mark the document accordingly, rather than leaving the section blank. Your reader will wonder whether the information is coming, or whether you just made a mistake.

The next rule is this:

2. Avoid repetition.

Each kind of information should be recorded in exactly one place. This makes documentation easier to use and *much* easier to change as it evolves. It also avoids confusion, because information that is repeated is often repeated in a slightly different form, and now the reader must wonder: Was the difference intentional? If so, what is the meaning of the difference? What information was the author trying to convey to me that I am not picking up?

Here in the wondrous age of online hypertext documents and Web media, there is nothing wrong with providing multiple access routes or entry points to a section (or Web page) that contains a specific kind of information. But the information itself should be stored in a single place for ease of change and consistent presentation to the user.

Now, expressing the same idea in different forms is often useful for achieving a thorough understanding. You could make the case that the whole concept of architectural views—see Section 1.2—flows from exactly this concept. But it should be a goal that information should never be repeated, or almost repeated, verbatim.

3. Avoid unintentional ambiguity.

In some sense, the point of architecture is to be ambiguous. A primary reason architecture is useful is because it suppresses or defers the plethora of details that are necessary to resolve before bringing a system to the field. The architecture is therefore ambiguous, one might argue, with respect to these suppressed details. But this is planned ambiguity. Even though an architecture may be brought to fruition by any of a number of different implementations, as long as those implementations comply with the architecture, they are all correct. Unplanned ambiguity is when documentation can be interpreted in more than one way, and at least one of those ways is incorrect. A well-defined notation with precise semantics goes a long way toward eliminating whole classes of linguistic ambiguity from a document. This is one area where architecture description languages help a great deal, but using a formal language isn't always necessary. Just adopting a set of notational conventions and then avoiding unplanned repetition (especially the "almost-alike" repetition mentioned previously) will help eliminate whole classes of ambiguity.

One of the greatest sources of ambiguity in architecture documentation are those ubiquitous box-and-line diagrams that people often draw on whiteboards or backs of napkins. While not a bad starting point, these diagrams are certainly not architectures. For one thing, the behavior of the components is not defined, and this (as we shall see) is a crucial part of the architecture. But beyond that, most of these diagrams suffer from ambiguity with respect to the component and connector types. Are the boxes supposed to be modules, objects, classes, processes, functions, procedures, processors, or something else? Do the arrows mean submodule, inheritance, synchronization, exclusion, calls, uses, data flow, processor migration, or something else?

We have two things to say about box-and-line diagrams purporting to be architectures. First, don't be guilty of drawing one and claiming it's anything more than a start at an architecture. Second, if you see one, ask its author what the boxes mean and what *precisely* the arrows connote. The result is almost always illuminating, even if the only thing illuminated is the owner's confusion, but it is often entertaining as well.

4. Use a standard organization.

Each document should conform to a standard, planned organization scheme, and this scheme should be made known to the reader. A standard organization offers many benefits. It helps the reader navigate the document and find specific information quickly (and so this is also related to the write-for-the-reader rule). But it also helps the writer of the document. It helps plan and organize the contents, and it reveals instantly what work remains to be done by the number of sections that are still blank or contain "TBD" marks. Finally, a standard organization embodies completeness rules for the information in the document; the sections of the document constitute the set of important aspects that need to be conveyed by the document. Hence, the standard organization can form the basis for a first-order validation check of the document at review time.

5. Record rationale.

If you are documenting the results of decisions, record the decisions you eschewed and say why. Next year (or next month) when those decisions come under scrutiny or pressure to change, you will find yourself revisiting the same arguments and wondering why you didn't take some other path. Recording rationale will save you enormous time in the long run, although it requires discipline to record in the heat of the moment.

6. Keep it current.

Documentation that is incomplete, out of date, does not reflect truth, and does not obey its own rules for form and internal consistency will not be used. Documentation that is kept current and accurate will be used. The reason is that, backed up by high-quality documentation, questions about the software can be most easily and most efficiently answered by referring the questioner to the appropriate document. If the documentation is somehow inadequate to answer the question, then it needs to be fixed. Updating it and *then* referring the questioner to it will deliver a strong message that the documentation is the final authoritative source for information.

7. Review documentation for fitness of purpose.

Only the intended users of a document will be able to tell you if it contains the right information presented in right way. Enlist their aid. Before a document is released, have it reviewed by representatives of the community or communities for whom it was written. The Chapter "Validate" covers this topic in more detail.

Any software documentation should obey these seven rules, including software architecture documentation. The chapter on "Views" will present other criteria that apply specifically to architecture documentation; "Beyond Views" will prescribe specific documentation for architecture that transcends views; and "Validate" will show how to validate software architecture documentation to make sure it is of high quality and utility.

1.2 Views

A software architecture is a complex entity that has thus far avoided description in a simple one-dimensional fashion. The analogy with building architecture, if not taken too far, proves illuminating. There is no single rendition of a building architecture. Instead, there are many: The room layouts, the elevation drawings, the electrical diagrams, the plumbing diagrams, the ventilation diagrams, the traffic patterns, the sunlight and passive solar views, the security system plans, and many others. Which of these views *is* the architecture? None of them. Which views *convey* the architecture? All of them.

So it is with software architecture. As long as ago as 1974, Parnas observed that software comprises many structures, which he defined as a partial description of a system showing it as a collection of parts and showing some relations between the parts [Parnas 74]. This definition largely survives in architecture papers today. Parnas identified several structures prevalent in software. A few were fairly specific to operating systems (such as the structure that defines what process owns what memory segment) but others are more generic and broadly applicable. These include the module structure (the units are work assignments, the relation is "is a part of" or "shares part of the same secret as"), the uses structure (the units are programs, and the relation is "depends on the correctness of"), and the process structure (the units are processes, and the relation is "gives work to").

More recently, Philippe Kruchten of the Rational Corporation wrote a compelling paper describing four main views of software architecture that can be used to great advantage in system-building, plus a distinguished fifth view that ties the other four together—the so-called "four plus one" approach to architecture [Kruchten 95]. The *logical view* primarily supports behavioral requirements—the services the system should provide to its end users. Designers decompose the system into a set of key abstractions, taken mainly from the problem domain. These abstractions are objects or object classes that exploit the principles of abstraction, encapsulation, and inheritance. In addition to aiding functional analysis, decomposition identifies mechanisms and design elements that are common across the system.

The *process view* takes into account some requirements such as performance and system availability. It addresses concurrency and distribution, system integrity, and fault tolerance. The process view also specifies which thread of control executes each operation of each class identified in the logical view. The process view can be seen as a set of independently executing logical networks of communicating programs ("processes") that are distributed across a set of hardware resources, which in turn are connected by a bus or local area network or wide area network. The *development view* focuses on the organization of the actual software modules in the software-development environment. The units of this view are small chunks of software—program libraries or subsystems—that can be developed by one or more developers. The development view supports the allocation of requirements and work to teams, and supports cost evaluation, planning, monitoring of project progress, and reasoning about software reuse, portability, and security. It is the basis for establishing a line of product. The *physical view* takes into account the system's requirements such as system availability, reliability (fault tolerance), performance (throughput), and scalability. This view maps the various elements identified in the logical, process, and development views—networks, processes, tasks, and objects—onto the processing nodes. Finally, Kruchten prescribes using a small subset of important scenarios—instances of use cases—to show that the elements of the four views work together seamlessly. This is the "plus one" view, redundant with the others but serving a distinct purpose.

At about the same time, Dilip Soni, Robert Nord, and Christine Hofmeister of Siemens Corporate Research made a similar observation about views of architecture they observed in use in industrial practice [Soni 95]. They wrote that the conceptual view describes the system in terms of its major design elements and the relationships among them. The module interconnection view encompasses two orthogonal structures: functional decomposition and *layers.* The *execution view* describes the dynamic structure of a system. Finally, the *code view* describes how the source code, binaries, and libraries are organized in the development environment.

In 1995, the authors of the seminal book on design patterns (Gamma, Helms, Johnson, and Vlissides) wrote the following:

> *An object-oriented program's runtime structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's runtime structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understanding one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa* [Gamma 95].

Exactly right. Like electrical and plumbing diagrams, each view of a software architecture is used for a different purpose, and often by different stakeholders.[1] The number of possible views of an architecture is limitless, but in practice there is a manageable number of views that are used in practice. Each is like a projection along one dimension of a multi-dimensional object that is so complex it does not exist in our sensory universe. Like building blueprints, architectural views are both descriptive—they describe the architecture as it was built—and also prescriptive—they represent constraints on the builders and are designed the way they are

[1] A stakeholder is someone who has a vested interest in the architecture.

to achieve particular qualities in the resulting system (such as performance or modifiability) or economies in the development project (such as time to market).

The fact that an architecture does not holistically exist and we cannot fully grasp it is the cause of no small consternation. We feel it is somehow inadequate to see it only through discrete views that may or may not relate to each other in any straightforward way. It makes us feel like the blind men groping the elephant. And yet, the essence of architecture is the suppression of information not necessary to the task at hand, and so it is somehow fitting that the very nature of architecture is such that it never presents its whole self to us, but only a facet or two at a time.

We use the concept of views to give us the most fundamental principle of architecture documentation, which we state as an axiom:

> *Documenting an architecture is primarily a matter of documenting the relevant views of that architecture, plus recording information that applies to more than one view.*

There are many views of a software architecture that are possible, more than the ones introduced above. Documenting views will be discussed in the chapter entitled "The View Zoo". Documenting information that transcends views will be covered in the chapter entitled "Documentation Beyond Views."

1.3 Uses of Architecture Documentation

The axiom stated in the preceding section prescribes documenting the relevant views of the architecture. Which views are relevant? The answer, of course, is "it depends." In particular, what you put into software architecture documentation depends a great deal on what you wish to get out of it. So the question becomes: "How do you expect your documentation to be used?" The answers will determine the form and content that your documentation should take.

We'll begin by thinking about the uses of architecture corresponding to the times in a project's lifetime when the various roles come into play.

A vehicle for communicating the system's design to interested stakeholders at each stage of its evolution.

This perspective on architecture is forward-looking, involving steps of creation and refinement. Stakeholders include those involved in managing the project, as well as "consumers" of the architecture that must write code to carry it out, or design systems that must be compatible with it. Specific uses in this category include the following:

- For downstream designers and implementors, the architecture provides their "marching orders." The architecture establishes inviolable constraints (plus exploitable freedoms) on downstream development activities.
- For testers and integrators, the architecture dictates the correct black-box behavior of the pieces that must fit together.
- For technical managers, architecture provides the basis for forming development teams corresponding to the work assignments identified.
- For project managers, architecture serves as the basis for a work breakdown structure, planning, allocation of project resources, and tracking of progress by the various teams.
- For designers of other systems with which this one must interoperate, the architecture defines the set of operations provided and required, and the protocols for their operation, that allows the interoperation to take place.

A basis for performing up-front analysis to validate (or uncover deficiencies in) architectural design decisions and refine or alter those decisions where necessary.

This perspective on architecture is, in some sense, inward-looking. It involves making prospective architectural decisions and then projecting the effect of those decisions on the system or systems that the architecture is driving. Where the effect is unacceptable, the relevant decisions are re-thought, and the process repeats. This process occurs in tight cycles (most architects project the effect of each of their decisions) and in large cycles (in which large groups of decisions, perhaps even the entire architecture, are subjected to formal validation). In particular, architecture provides the following:

- For the architect and requirements engineers who represent the customer(s), architecture is a forum for negotiating and making trade-offs among competing requirements.
- For the architect and component designers, architecture is a vehicle for arbitrating resource contention and establishing performance and other kinds of run-time resource consumption budgets.
- For those wanting to develop using vendor-provided products from the commercial marketplace, the architecture establishes the possibilities for commercial off-the-shelf (COTS) component integration by setting system and component boundaries and establishing requirements for the required behavior and quality properties of those components.
- For those interested in the ability of the design to meet the system's quality objectives, the architecture serves as the fodder for architectural evaluation methods such as the Software Architecture Analysis Method [Kazman 96] and the Architecture Tradeoff Analysis Method (ATAM^{SM}) [SEI 00] and Software Performance Engineering (SPE) [Smith 90] as well as less ambitious (and less effective) activities such as unfocused design walkthroughs.

- For performance engineers, architecture provides the formal model that drives analytical tools such as rate monotonic schedulers, simulations and simulation generators, theorem provers and model checking verifiers.
- For product line managers, the architecture determines whether a potential new member of a product family is in or out of scope, and if out, by how much.

The first artifact used to achieve system understanding.

This perspective on architecture is reverse-looking. It refers to cases in which the system has been built and deployed, and now the time has come to make a change to it or to extract resources from it for use elsewhere. Architecture mining and recovery fall into this category, as do routine maintenance activities. In particular, architecture serves the following roles:

- For technical mangers, architecture is basis for conformance checking, for assurance that implementations have in fact been faithful to the architectural prescriptions.
- For maintainers, architecture is a starting point for maintenance activities, revealing the areas a prospective change will affect.
- For new project members, the architecture is usually the first artifact for familiarization with a system's design.
- For those inheriting the job of architect after the previous architect's untimely departure, the architecture is the artifact that (if properly documented) preserves that architect's knowledge and rationale.
- For re-engineers, architecture is the often first artifact recovered from a program understanding activity or (in the event that the architecture is known or has already been recovered) the artifact that drives program understanding activities at component granularities.

It should be clear from this discussion that architecture documentation is both prescriptive and descriptive. That is, it prescribes what should be true, and it describes what is true, about a system's design. In that sense, the same documentation serves both purposes, and rightly so. If the "build-as" documentation differs from the "as-built" documentation, then clearly there was a breakdown in the development process.

However, the best architectural documentation for, say, performance analysis may well be different from the best architectural documentation we would wish to hand to a module implmementor. After we introduce architectural views and other architectural documentation, we will return to the roles of architecture and discuss what documentation strategies are well suited to each role.

2 The Layered View

2.1 Introduction

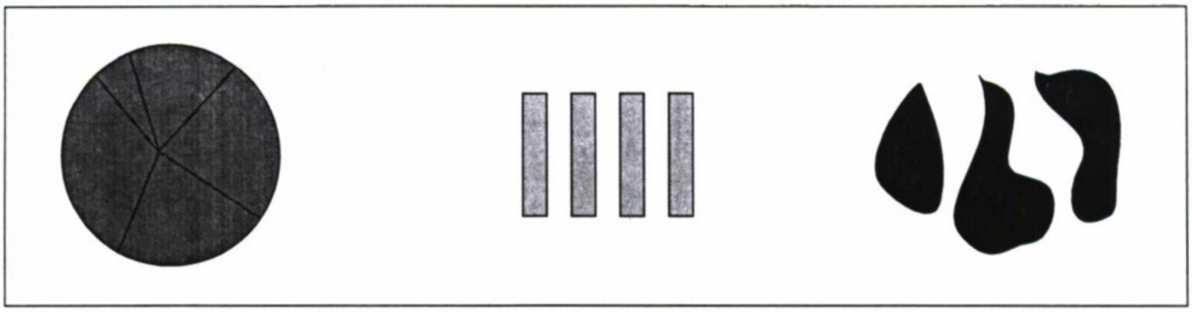
The layered view of architecture, shown with a layer diagram, is one of the most commonly used views in software architecture. It is also poorly defined and often misunderstood. Because true layered systems have good properties of modifiability and portability, architects have incentive to show their systems as layered, even if they are not.

Layering, like all architectural structures, reflects a division of the software into units. In this case, the units are layers; each layer represents a *virtual machine*. A virtual machine is a collection of software that together provides a cohesive set of services that other software can utilize without knowing how those services are implemented. Programming languages such as C++ meet this definition: Although the ultimate result is machine code that executes on one or more processors somewhere, we regard the instruction set provided by the language as the ultimate *lingua franca* of our program. We forget, happily, that without other virtual machines (the operating system, the microcode, the hardware) underneath C++, our program would just be a collection of alphanumeric characters that wouldn't do anything.

Any unit of software that has a public *interface* provides a set of services, but does not necessarily constitute a virtual machine. The set of services must be cohesive with respect to some criterion. The services might all appeal to a particular area of expertise (such as mathematics or network communication). Or they might be native to some application area (such as maintaining bank accounts or navigating an aircraft). The goal of layering is to define virtual machines that are small enough to be well understood, but comprehensive enough so that likely changes will affect only a single layer.

To recap, layers partition the software, and each partition constitutes a virtual machine (with a public interface) that provides a cohesive set of services. But that's not all. The following figure (which is intentionally vague about what the units are and how they interact with each other) shows three divisions of software—and you'll have to take our word that each division is a virtual machine—but none of them constitutes a layering. What's missing?

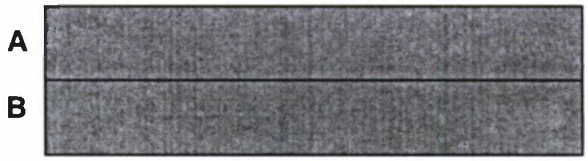
Layering has one more fundamental property: The virtual machines are created to interact with each other according to a strict ordering relation. Herein lies the conceptual heart of layers. If (A,B) is in this relation, we say "layer B is beneath layer A", and that means either, or both, of the following:



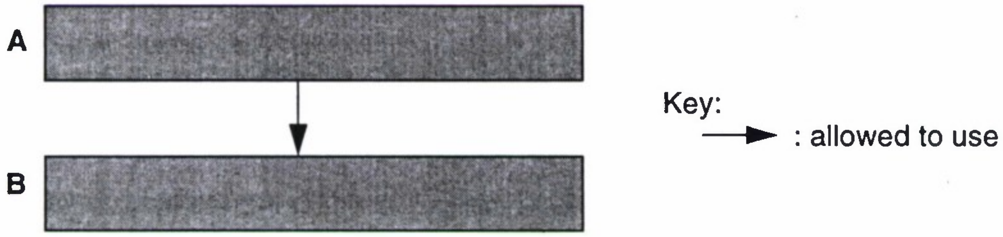
1. "The implementation of layer A is allowed to use any of the public facilities of the virtual machine provided by layer B."
2. "The public facilities in layer B are allowed to be used by the software in layer A."

By "use" and "used" we mean the *uses* relation as defined by Parnas in 1979 [Parnas 79]: A unit of software A is said to *use* unit B if A's correctness depends upon a correct implementation of B being present.

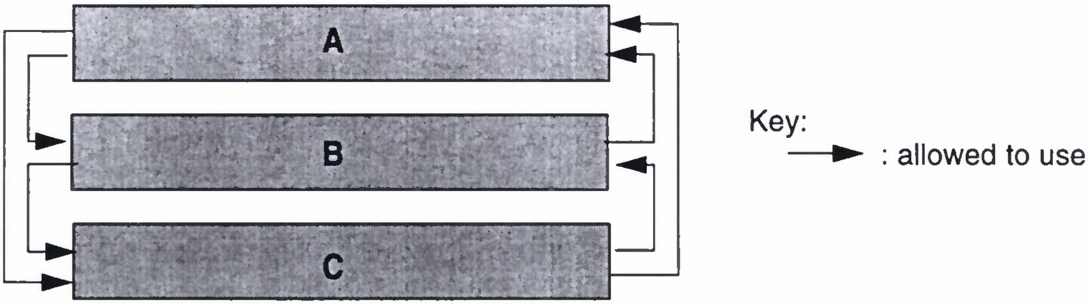
We draw the layering relation like this:



Layering is thus one of the few architectural views in which connection among components is shown by geometric adjacency and not some explicit symbology such as an arrow, although arrows can be used, like this:



There are some loopholes in the definition. If A is implemented using B, is it implemented using *only* B? Maybe; maybe not. Some layering schemes allow a layer to use the public facilities of *any* lower layer, not just the nearest lower layer. Others are more restricted. *But no valid layering scheme allows a layer to use, without restriction, the facilities of a higher layer.* (See the sidebar "Upwardly Mobile Software") Allowing unrestricted upward usage destroys the desirable properties that layering brings to an architecture; this will be discussed shortly. Usage in layers generally flows downward. A small number of well-defined special cases may be permitted, but these should be few and regarded as exceptions to the rule. Hence, the following architectural view resembles a layering, but is not:



Figures like the one above are why layers have been a source of ambiguity for so long, for architects have been calling such things layered when they are not. There is more to layers than the ability to draw separate parts on top of each other.

Because the ordering relationship among layers has to do with "implementation allowed to use," the lower the layer the less facilities are available to it apart from the native hardware itself. That is, the "world view" of lower layers tends to be smaller and more focused on the computing platforms. More knowledge about the operating environment tends to be a part of their design. Lower layers tend to be built using knowledge of the computers, communications channels, distribution mechanisms, process dispatchers, and the like. These areas of expertise are largely independent of the particular application that runs on them (meaning they will not need to be modified if the application changes). Viewed from a higher layer, these low layers provide a virtual machine that may be distributed and that provides facilities to handle all of the communication and distribution issues for programs that run on it.

Conversely, higher layers tend to be more independent of the hardware; they can afford to be, because the existence of the lower layers has given them that freedom. This means they are not likely to change should there be a change to the computing platform or environment; they can afford to be concerned only with details more native to the application.

Some other observations about layers:

- Layers cannot be derived by examining source code. The source code will disclose what actually uses what, but the relation in layers is "allowed to use." As a trivial example, you can tell by code inspection that a "double" operation was implemented using multiplication by 2, but you cannot tell from the code
  - whether it would have been equally acceptable to implement double(x) by adding x to itself or by performing a binary left shift—that is, what double(x) was allowed to use
  - whether addition, double, and multiplication are in the same or different layers

- A layer may provide services that are not actually used by other software. This usually occurs when the layer was designed to be more general than is strictly necessary for the application in which it finds itself. This in turn often occurs when the layer is imported from some other application or purchased as a commercial product. A service that is never used may needlessly consume some run-time resource (such as memory to store the unused code, or a thread that is never launched). If these resources are in short supply, then a sophisticated compile-link-load facility that eliminates unused code will be helpful. Lacking this, the situation can be viewed as a tradeoff between efficiency and portability.
- In many layered systems there will be situations in which software of a lower layer will have to use (and therefore must be "allowed to use") software in a higher layer and these usages will have to be accommodated by the architecture. In other cases, software in a very high layer might be required to directly use software in a very low layer where normally only next-lower-layer uses are allowed. The layer diagram (or an accompanying document) will have to show these exceptions (called *bridging*). If there are many of these, it is a sign of a poorly structured system (at least with respect to the portability and modifiability goals that layering helps to achieve). Systems with upward usages are not, strictly according to the definition, layered. However, in such cases, the layered view represents a close approximation to reality, and also conveys the ideal design that the architect's vision was trying to achieve.
- Each layer could be implemented using a different style. Take for example a system defined using a pipes and filters style of architecture. The topmost layer would contain the modules corresponding to the pipes and filters of the system. The data passing through the pipes could be so large that it makes sense to pass the data using shared memory. Therefore, the next layer down could contain the modules corresponding to the components of a shared memory style. The system defined in the pipe and filter style runs on this virtual machine, calling the shared memory capabilities to pass data between filters.

With this overview, we now complete our exposition of layers.

2.2 Elements/Relations/Properties of the Layered View

Elements

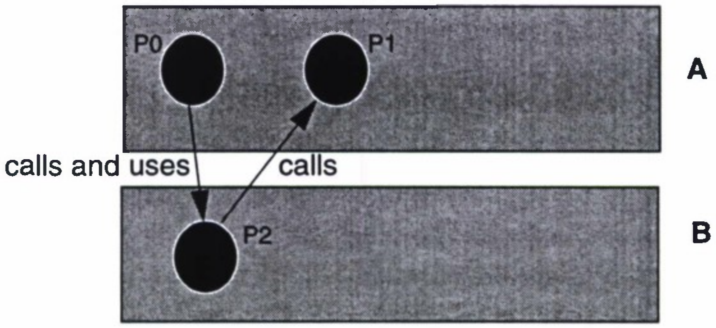
The elements of a layered diagram are *layers*. A layer is a collection of software units such as programs or modules that may be invoked or accessed. This definition admits many possibilities, from objects (or classes) down to assembly language subroutines and many things in between, such as a shared memory. A requirement is that the units have an interface by which their services can be triggered or initiated or accessed.

Relations

The relation among layers is "allowed to use." If two layers are related by this relation, a unit of software in the first is allowed to use any unit of software in the second. A unit of software A is said to use another unit B if A's correctness depends on B being correct as well. The allowed-to-use relation is usually anti-symmetric. That is, if (A,B) is in it then (B,A) probably should not be. (The layout is always anti-symmetric: If B is underneath A, then A cannot be underneath B.)

The relation among layers appears to resemble, but is decidedly not, the simple calls relation provided by the family of imperative programming languages. Here's why:

- A program P1 can use program P2 without calling it. P1 may assume, for example, that P2 has left a shared device in a usable state when it finished with it. Or P1 may expect P2 to leave a computed result that it needs in a shared variable. Or P1 may be a process that sleeps until P2 signals an event to awaken it. Thus, just because P1 resides in a layer over P2, we cannot conclude that P1 under any circumstances calls P2.
- A program P1 might call program P2 but not use it. If P2 is an error handler that P1 calls when it detects an error of some sort, P1 will usually not care in the least what P2 actually does. The figure below shows an example: P0 calls P1 to accomplish some work, and depends on its result. P0 thus uses P1, and their relationship is consistent with P1 residing in a layer beneath that of P0. Suppose P0 calls P1 incorrectly. Part of P1's specification is that in case of error, it calls a program whose name is passed to it by its client.[1] Once P1 calls that program, it has satisfied its specification. In this case, P1 calls (but does not use) P2, which makes it perfectly acceptable for P2 to reside in a layer above that of P1. This is what we want, for P2 is likely privy to the same knowledge about what was intended as P0; it makes sense that they are in the same layer (virtual machine) as each other.



[1] Or perhaps it calls a program whose name was bound by parameter at system-generation time, or a program whose name it looks up via some name server. As long as the name of the error handler is not hard-coded into P1, P1's layer remains portable.

Properties

A layer has the following properties:

Cohesion. It provides a cohesive set of services, meaning that the services as a group would likely be useful (as a group) in some other context than the one in which they were developed.

For example, suppose program P1 is allowed to use program P2. Should P2 be in a lower layer than P1, or should they be in the same layer? Layers are not a function of just who-uses-what, but are the result of a conscious design decision that allocates software to layers based on considerations such as coupling, cohesion, and likelihood of changes. In general, P1 and P2 should be in the same layer if they are likely to be ported to a new application together, or if together they provide different aspects of the same virtual machine to a usage community. This is an operational definition of cohesion. These questions cannot be answered using a code analyzer, but only experience and domain knowledge.

Interface. It provides a set of public interface facilities that may be invoked or accessed by other software. In cases where the layer is widely used across many kinds of systems and organizations, its interface may well be a public standard.

2.3 Semantics and Well-Formedness

The primary rule of well-formedness is that the geometry of the layer diagram specifies a relation that is an ordering among its elements, the layers. If (A,B) is in the relation, then software in A is allowed to use software in B.

A layer diagram must be accompanied by the following elaborating documentation:

Key. An explanation of how the allowed-to-use relation is carried out by the geometry of the figure. In particular, it should answer the following questions:

- Is a layer allowed to use only the layer below, any lower layer, or some other?
- Is software permitted to use other software in the same layer? (Usually the answer is "yes" because doing so has no ill effect on the modifiability achieved by the layers.)
- Are the relative sizes of layers significant? Unless otherwise specified, they are not.
- Color is often used to impart meaning about certain layers. If so, a key should be included to explain the significance of color.
- It should be made clear that no upward usage is assumed to be allowed except where explicitly described.

Layer interfaces. A document that shows what facilities constitute the public interface to each layer. Since layers consists of public and private software, an implementor of software in an upper layer must know what facilities are available from the lower layer. A layer interface often consists of all of the public interfaces for the units of software it contains; therefore, the interface to the layer may be documented under the other architectural view that describes those units. In this case, the layer interface document is a pointer to that other view.

Layer catalog. A document that assigns each unit of software to exactly one layer. Layer diagrams will typically label the layers with descriptive (but vague) names like "network communications layer" or "business rules layer" but a catalog is needed that lists the complete contents of every layer.

Layer portability guide. A document that describes the changes that can be made to each layer without affecting other layers.

Exceptions. A document that explains exceptions, if any, to the usage rules implied by the geometry. Exceptions may be upward (allowing something in a lower layer to use something above) or downward (either prohibiting a specific usage otherwise allowed by the geometry, or by allowing downward usage that "skips" intermediate layers normally required). Exceptions should be precisely described.

2.4 What It's For and What It's Not For

Layers help to bring quality attributes of modifiability and portability to a software system. A change to a lower layer that does not affect its interface will require a change to no higher layer. For example, any operating system that meets the Posix standard may be freely substituted without change to application-level software. A change to a higher layer that does not affect what facilities it requires from lower layers will not affect any lower layer. In general, changes to a layered software system that affect no interface are confined to a single layer. Thus, layers define units of reuse and portability.

Layers are part of the blueprint role that architecture plays for constructing the system. Knowing the layers in which their software resides, developers know what services they can rely on in the coding environment. Layers may define work assignments for development teams (although not always).

Layers are part of the communication role played by architecture. In a large system, the number of dependencies among modules expands rapidly. Organizing the software into layers with interfaces is an important tool to manage complexity and communicate the structure to developers.

Layers help with the analysis role played by architecture: They can be used for analyzing the impact of changes to the design.

Finally, layers (with their allowed-to-use relation) are a precursor to tracking the actual *uses* relation that emerges when coding is complete. The uses relation is the key to fielding subsets of the system in an incremental fashion, which is a rich and powerful technique for coping with schedule slips (and thus delivering a functional subset, as opposed to nothing, at the deadline) and controlling complexity (by adding functionality in a methodical fashion, rather than building everything and hoping it works). A subset is defined as follows: If you want program A in the subset, then take the transitive closure of its uses relation. That is, the subset must include all programs that A uses, and all programs that they use, and so forth. (Programs called but not used must be included also, but only as stubs.) Nothing else is required.

An unrestricted uses relation in which every program uses every other program results in every program being a member of every subset, which of course means that no proper subset is possible at all. Layering is meant to head off this possibility, paving the way for subsets consisting of only a small number of programs, which means that a working system that does something useful can be accomplished in a short time and is a very manageable task. This is a strong reason why upward uses are strongly discouraged if not disallowed: upward uses, and their transitive closure, tend to make subsets large and unwieldy tangles of interrelated programs that are far too complex to be workable.

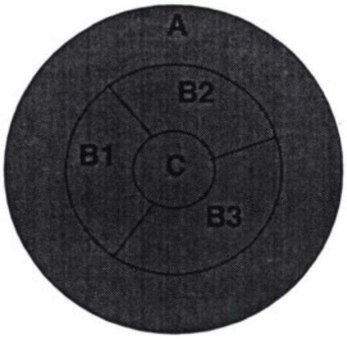
2.5 Notations

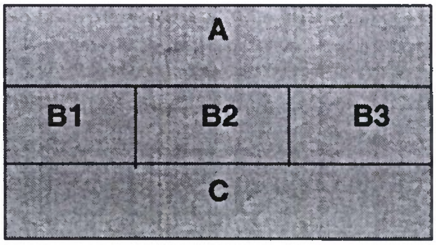
Informal notations

Stack. Layers are almost always drawn as a stack of rectangles atop each other. The allowed-to-use relation is denoted by geometric adjacency, or sometimes by an arrow.

Rings. The most common variation is to show layers as a set of concentric circles or rings. The innermost ring corresponds to the lowest layer. The outermost ring corresponds to the highest layer. A ring may be subdivided into sectors, meaning the same thing as the corresponding layer being subdivided into parts. (See Section 2.6 for more about segmented layers.)

There is no semantic difference between a layer diagram that uses a "stack of rectangles" paradigm and one that uses the "ring" paradigm—except in one case. In the figure on the left below, assume that segments in the same ring that touch each other are allowed to use each other. The corresponding rule in the "stack" picture would be that segments in the same layer that touch each other are allowed to use each other. There is no way to "unfold" the ring picture to produce a stack picture (such as the one on the right) with exactly the same meaning, because circular arrangements allow more adjacencies than linear arrangements. (In the figure on the right, B1 and B3 are separate, whereas in the figure on the left they are adjacent.) Cases like this are the only ones where a ring picture can show a geometric adjacency that a stack picture cannot.





Thick edges. Sometimes the rectangles in a stack are shown with thick horizontal edges denoting the interface to the layer. This is intended to convey the restriction that inter-layer usage only occurs via interface facilities, and not directly to any layer's "internals."

Size and color. Sometimes layers are colored to denote which team is responsible for them or some other distinguishing feature. Size is sometimes used to give a (vague) idea of the relative size of the software constituting the various layers. Size and color should be explained in the key accompanying the layer diagram.

Formal notations

UML. UML [Booch 98] has no built-in primitive corresponding to a layer. However, simple (non-segmented) layers can be represented in UML using packages. A package is a general-purpose mechanism for organizing elements into groups. UML has pre-defined kinds of packages for systems and subsystems. We can introduce an additional one for layers by defining it as a stereotype of package. Stereotypes allow us to tailor the definition of a UML element, typically by adding additional constraints and/or changing the visual notation. A layer can be shown as a UML package with the constraints that it groups units of software together and that the dependency between packages is "allowed to use." We can designate a layer using the package notation with the stereotype name <<layer>> preceding the name of the layer or introduce a new visual form, such as a shaded rectangle.

The allowed-to-use relation can be represented as a stereotype of the UML access dependency, one of the existing dependencies between packages. This dependency permits the elements of one package to reference the elements of another package. More precisely:

- An element within a package is allowed to access other elements within the package.

- If a package accesses another package, then all elements defined with public visibility in the accessed package are visible within the importing package.
- Access dependencies are not transitive. If package 1 can access package 2 and package 2 can access package 3, it does not necessarily follow that package 1 can access package 3.

We add the additional constraint that the defined dependency is anti-symmetric. This allows us to represent a strict or partial ordering relation among layers.

Packages can also be used to represent a layers' interfaces. Packages allow three kinds of interface descriptions:

1. The sum of all element interfaces. By default, everything within a package is visible.
2. A subset of element interfaces. Packages have a mechanism to define the visibility of the elements it contains. Therefore, a layer interface could be a subset of the interfaces of its elements.
3. A separate layer interface. For this case, one could define the interface as a collection of classes within the package (using the façade pattern, for example) making them visible and hiding everything else.

Packages therefore serve as a way to fulfill all of the following documentation obligations concerning layers: layer diagram, key, exceptions, layer catalog, and layer interfaces. The portability guide could be given as an annotation associated with each package.

Open issues that arise when using UML to represent layers include the following:

- Elements can only be owned by a single package. If an element needs to be a member of a layer *and* a subsystem, packages cannot be used to represent both.
- It is not clear how to represent callbacks with UML. Callbacks are a common method of interaction between software in different layers; see "Upwardly Mobile Software."
- It is not clear how best to represent segmented layers in UML. Segmented layers are discussed in the next section.

2.6 Variations

Common variations to the layered diagram include the following:

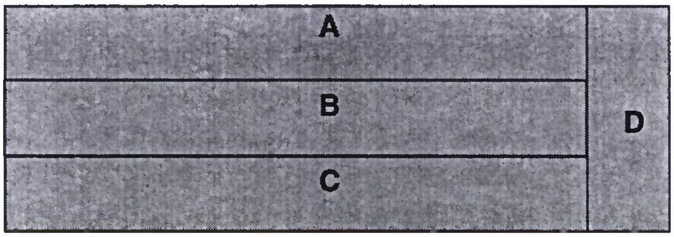
1. Reach. The most common variation in layers are the conventions that determine the extent of each layer's allowed-to-use relation. Typically these conventions allow software in a layer to use software in
   a. the closest lower layer

   b. any lower layer

   c. any adjacent layer, above or below

   or some other exigency. In addition, the convention must specify whether software in a layer is allowed to use other software at the same level.

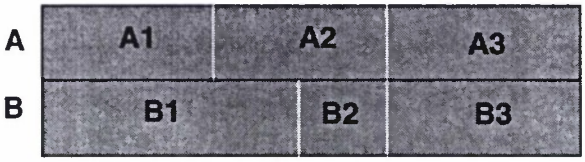
2. Layers with a sidecar. Many architectures called "layered" look something like the following:



   This could mean one of two things. Either software in D can use software in A, B, or C. Or, software in A, B, or C can use software in D. (Technically, the diagram might mean that both are true, although this would arguably be a poor architecture.) It is incumbent upon the creator of the diagram to say which is the case.

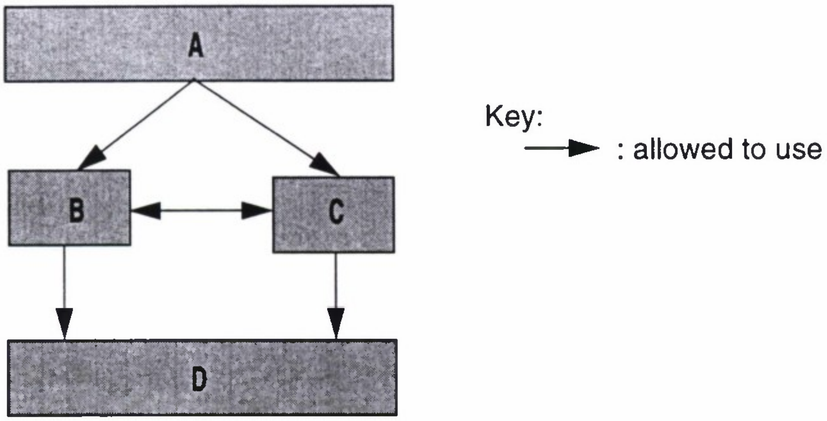
   A variation like this only makes sense when the usage rules in the main stack are single-level: that is, when A can use only B, and nothing below. Otherwise, D could simply be made the topmost (or bottom-most, depending on the case) layer in the main stack, and the "sidecar" geometry would be completely unnecessary.

3. Segmented layers. Sometimes layers are divided into segments denoting some finer-grained decomposition of the software. Often this occurs when there is some pre-existing set of units (such as imported components or components from separate teams) that share the same allowed-to-use relation. When this happens, it is incumbent upon the creator of the diagram to say what usage rules are in effect among the segments. May they use each other? If the next-lower layer is also divided into segments, does geometric adjacency determine usage or may any segment in one layer use any segment in the next lower layer? That is, in the figure below...

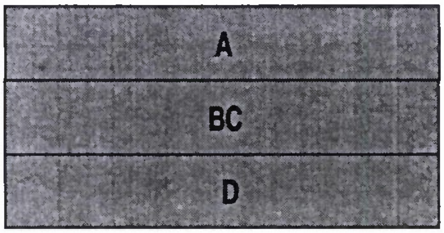


   ...is software in A1 allowed to use software in B2 and B3, or only software in B1? And issoftware in A2 allowed to use software in B1 and B2 but not B3? And may A1, A2, and A3 use each other? These relations must be spelled out.

Segmented layers essentially make the allowed-to-use relation a partial ordering of the elements. The one below specifies that A is allowed to use B and C, which are in turn allowed to use D *and each other.*



From the strict point of view of layers, the diagram above is completely equivalent to this one:



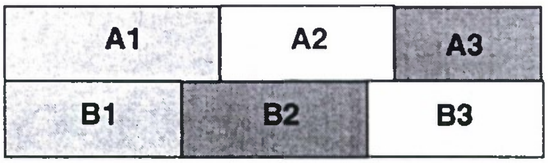
where layer BC is the union of the contents of layers B and C. That's because the allowed-to-use relation depicted by the two diagrams is the same. The decomposition of the middle layer into B and C brings additional information to the diagram that has nothing to do with layering—perhaps B and C have been developed by separate teams, or represent separate modules, or will run on different processors.

4. Layering through inheritance. This is a variation that is consistent with the basic model of layers presented here, but interesting in the object-oriented world. Layering through inheritance occurs when a base class is in a lower layer and a subclass that inherits from the interface specified by the base class is in a higher layer. The subclass certainly uses (and hence must be allowed to use) the base class, for its correctness depends on inheriting the correct attributes. The base class forms part of a virtual machine that can be used to create more application-specific instances of the class.

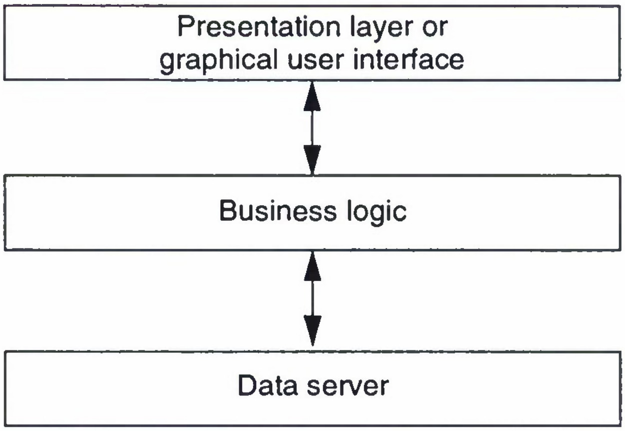
2.7 Confusions

Layer diagrams are often confused with other architectural views when information orthogonal to the allowed-to-use relation is introduced without conscious decision. In particular:

1. Work assignments. There is a tendency to regard layers as identical to work assignments. A layer may in fact be assigned as a unit of work, but it is not necessarily always so. One team's work assignment could span all or part of one or more layers. Segmented layers are often introduced to show work assignments. If a work assignment spans layers, colors or fill patterns are often used, as in the following:



2. Tiers. Layers are very often confused with the tiers in an n-tier client-server architecture, such as shown in the following figure:



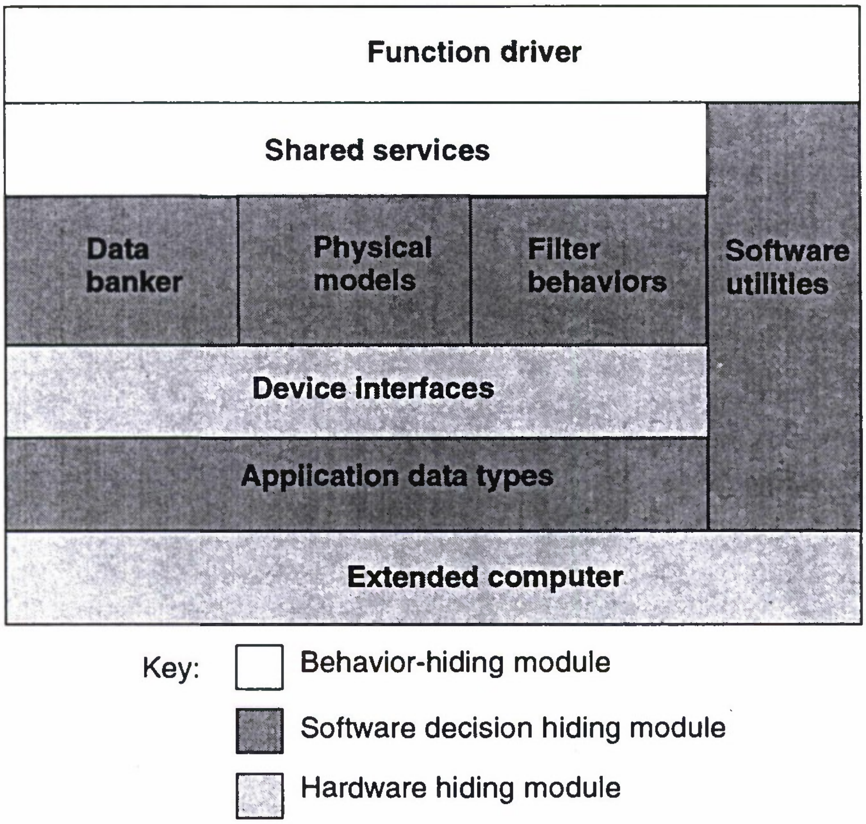
Despite the fact that this looks like a layer diagram (and the topmost element even uses the "L"-word in its name) diagrams such as this express concerns very different from layers. Allocation to machines in a distributed environment, data flow among elements, and the presence and utilization of communication channels all tend to be expressed in tier pictures, and these are indiscernible in layer diagrams. And notice the two-way arrows. Whatever relations are being expressed here (and as always, a key should tell us), they're bi-directional (symmetric) and we know that's bad news in a layer diagram.

Further, assignment of a unit of software to one of these elements is based on run-time efficiency of some sort: locality of processing, maintaining the ability to do useful work in case of network or server failure, not overloading the server, wise utilization of network bandwidth, etc.

Layers are not tiers. However, tiers do resemble layers. Each obeys an allowed-to-use relation; the presentation layer is implemented in terms of the virtual machine provided by the business logic, which in turn can use the data server. Not shown are the infrastructure layers that provide for inter-processor communication, operating system support, etc. Complicating the situation is the fact that because each tier usually resides on a separate machine, each tier might well have its own separate infrastructure.

3. Logical view. Layers are often confused with modules in the logical view, which will be discussed in a future version of this work. They have much in common. Both are collections of usable software services. Both have interfaces through which they require their clients to make use of their provided services. Both have private parts that are off-limits. Are they not the same?

   They may be the same, but do not have to be. Below is an example where they are not:



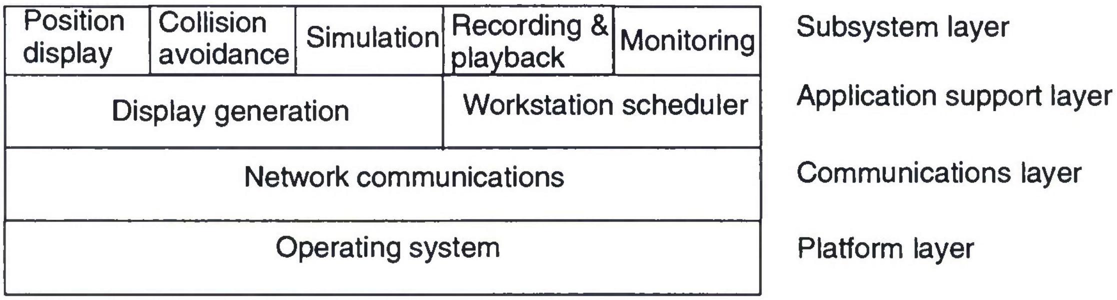
In this architecture, taken from [Bass 98 ch. 3], criteria for partitioning into modules was the encapsulation of likely changes. The shading of the elements denotes the coarsest-grain decomposition of the system into modules. In this system, layers corresponded to parts of modules. It's also easy to imagine a case where a module constitutes a part of a layer.

While modules and layers may not be the same, it is the case that the interfaces to layers consist of the interfaces to the modules (or parts of the modules) that constitute it.

2.8 Related Concepts

1. Subsystems. Layers cross conceptual paths with the concept of subsystem. A subsystem can be (like many concepts in this field) anything you want it to be, but it often describes a part of a system that (a) carries out some functionally cohesive subset of the overall system's mission; and (b) can be executed independently. Subsystems are subsets of the system that can be developed and deployed incrementally. An air traffic control system, for example, may be divided into several areas of capability:
   - interpreting radar data to display aircraft positions on screens
   - detecting aircraft that are about to violate separation constraints
   - running simulations for training
   - recording and playback for after-situation analysis and training
   - monitoring its own health and status

Each of these might reasonably be called a subsystem. Subsystems are often represented by using segmented layers. For example, if the air traffic control system looks like this:



...then a subsystem consists of a segment from the top layer, plus any segments of any lower layers it's allowed to use. A subset of the system, when shown in a layer diagram, is often called a slice or vertical slice.

2. Runtime relations. A layer diagram is just the first part of a three-part story. After you divide the software into units and lay out the allowable uses among those units, you must then establish what interaction mechanisms will actually carry out the uses. Finally, you must specify the actual uses and interactions among the units of software in the layers. The last two parts of the story are told using other architectural views, such as the style view, control-flow, data-flow, or the process views.

2.9 Vocabulary

application program interface (API) - A set of routines (functions, procedures, methods) and protocols. An API defines the boundary between layers.

interface - An interface defines the boundary between units of software. An interface defines the services provided and required by each unit of software.

layer - A virtual machine with a set of capabilities/services. These services can only be accessed through a layer's interface.

layer structure - A collection of layers where each layer provides services to the layer above and is allowed to use the services of the layer below.

uses - Procedure *A* is said to use procedure *B* if a correctly functioning procedure *B* must be present in order for procedure *A* to meet its requirements.

virtual machine - A collection of software that together provides a cohesive set of services that other software can utilize without knowing how those services are implemented.

bridging, layer bridging - An exception to the stated allowed-to-use relation indicated by the layers. Bridging usually occurs when a high layer uses a low layer directly without going through intermediate layers.

2.10 References

Bass 98 Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*. Reading, MA: Addison-Wesley Longman, 1998.

Booch 98 Booch, Jacobson, & Rumbaugh. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley Longman, Inc., 1998.

Buschmann 96 Buschmann, Meunier, Rohnert, Sommerlad, & Stal. *A System of Patterns: Pattern Oriented Software Architecture*. Chichester, NY: John Wiley & Sons, 1996.

Hofmeister 00 Hofmeister, Nord, & Soni. *Applied Software Architecture*. Reading, MA: Addison-Wesley, 2000.

Gamma 95 Gamma, E.; Helm, V.; Johnson, R.; & Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

Garlan 95 Garlan, David & Perry, Dewayne. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering 21*, 4 (April 1995).

Garlan 93 Garlan, D. & Shaw, M. "An Introduction to Software Architecture," in *Advances in Software Engineering and Knowledge Engineering,* vol. I. River Edge, NJ: World Scientific Publishing Company, 1993.

Kazman 96 Kazman, R.; Abowd, G.; Bass, L.; & Clements, P. "Scenario-Based Analysis of Software Architecture." *IEEE Software 13*, 6 (November 1996): 47-55.

Kruchten 95 Kruchten, P. "The 4+1 View Model of Architecture," *IEEE Software 12,* 6: 42-50, 1995.

Parnas 79 Parnas, D. L. "Designing Software for Ease of Extension and Contraction." *IEEE Transactions on Software Engineering SE-5*, 2: 128-137.

Parnas 76 Parnas, D. L. & Wurges, H. "Response to Undesired Events in Software Systems, " 437-446. *Proceedings of the Second International Conference on Software Engineering.* San Francisco, CA., October 13 - 15, 1976. New York: IEEE, 1976.

Parnas 74 Parnas, D. L. "On a 'Buzzword': 'Hierarchical Structure,' " *Proc. IFIP Congress,* 74: 336-390, 1974.

Perry 92 Perry, D.E. & Wolf, A.L. "Foundations for the Study of Software Architecture." Software Engineering Notes, *ACM SIGSOFT 17*, 4 (October 1992): 40-52.

SEI 00 Software Engineering Institute. "The Architecture Tradeoff Analysis Method" [online]. Software Engineering Institute, Carnegie Mellon University, 2000. Available WWW: URL <http://www.sei.cmu.edu/ata/ata_method.html>.

Shaw 96 Shaw, M & Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall, 1996.

Smith 90 Smith, C.U. *Performance Engineering of Software Systems.* The SEI Series in Software Engineering. Reading, MA: Addison-Wesley, 1990.

Soni 95 Soni, D.; Nord, R.; & Hofmeister, C. "Software Architecture in Industrial Applications, " 196-210. *Proceedings, International Conference on Software Engineering*. Seattle, WA, April 23 - 30, 1995. New York: ACM, 1995.

2.11 Sidebar: "Upwardly Mobile Software"

We have been downright pedantic about saying that upward uses invalidate layering. We made allowances for exceptions, but implied that too many of those would get you barred from the Software Architect's Hall of Fame.

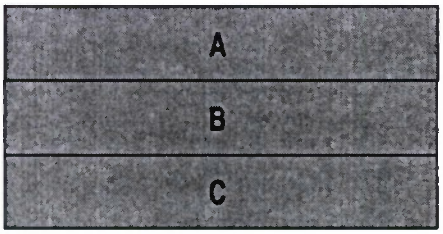
Seasoned designers, however, know that in many elegantly designed layered systems, all kinds of control and information flow upward along the chain of layers with no loss of portability, reusability, modifiability, or any of the other qualities associated with layers. In fact, one of the purposes of layers is to allow for the "bubbling up" of information to the units of software whose scope makes them the appropriate handlers of the information.

Error handling exemplifies this. In the now-classic stack-based error propagation scheme described by Parnas in 1976

> *...upon detecting an [error] in a hierarchically structured piece of software, the [error] is first reflected and control passed to the level where it originated. At this point it is either corrected or reflected still higher... At every level, either recovery is attempted or the UE is reported still higher* [Parnas 76].

The idea is that the software that caused the error is the best place to handle the error because the scope and information are available there to do so. When a layer is ported to some other application or environment, not only does the functionality transfer but also the ability to handle any errors that might be precipitated by that functionality. It makes a nice matching set.

Suppose we have a simple three-layer system:



Say that program Pa in A uses program Pb in B which uses program Pc in C. If Pc is called in a way that violates its specification, Pc needs a way to tell Pb "Hey! You called me incorrectly!" At that point, Pb can either recognize its own mistake and call Pc again (this time correctly) or take some other action, or Pb can realize that the error was due to the fact that it was called incorrectly (perhaps sent bad data) by Pa. In the latter case, it needs a way to tell Pa "Hey! You called me incorrectly!"

Callbacks are a mechanism to manifest the protestation. We do not want Pc written with knowledge about programs in B, or Pb written with knowledge about programs in A, since this would limit the portability of layers C and B. Therefore, the names of higher-level programs to call in case of error are passed downward as data. Then the specification for, say, Pb includes the promise that in case of error it will invoke the program whose name has been made available to it.

So there we have it: data and control flowing downward *and upward* in an elegant error-handling scheme that preserves the best qualities of layers. So much for our prohibition about upward uses. Right?

Wrong. Upward uses are still a bad idea, but the scheme we just described doesn't have any of those. It has upward data flow and upward invocation, but not uses. The reason is that once a program calls its error-handler, its obligation is discharged. The program does not *use* the error handler because its own correctness depends not a whit on what the error handler does.

While this may sound like a mere technicality, it is an important distinction. Uses is the relation that determines the ability to reuse and port a layer. Calls or "sends data to" is not. An architect needs to know the difference and needs to convey the precise meaning of the relations in his or her architectural documentation.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering end maintaining the data needed, end completing and reviewing the collection of information. Send comments regarding this burden estimate or eny other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, end to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

3. **AGENCY USE ONLY** (leave blank)

4. **REPORT DATE**
March 2000

5. **REPORT TYPE AND DATES COVERED**
Final

6. **TITLE AND SUBTITLE**
Software Architecture Documentation in Practice: Documenting Architectural Layers

7. **FUNDING NUMBERS**
C — F19628-95-C-0003

8. **AUTHOR(S)**
Felix Bachmann, Len Bass, Jeromy Carriere, Paul Clements, David Garland, James Ivers, Robert Nord, Reed Little

9. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

10. **PERFORMING ORGANIZATION REPORT NUMBER**
CMU/SEI-2000-SR-004

11. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

12. **SPONSORING/MONITORING AGENCY REPORT NUMBER**

13. **SUPPLEMENTARY NOTES**

12.a **DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified/Unlimited, DTIC, NTIS

12.b **DISTRIBUTION CODE**

13. **ABSTRACT** (maximum 200 words)

This report represents the first milestone of a work in progress. That work is a comprehensive handbook on how to produce high-quality documentation for software architectures. The handbook, tentatively entitled *Software Architecture Documentation in Practice*, will be published in mid- to late-2000 by Addison Wesley Longman as a book in the SEI series on software engineering. Aimed squarely at the practitioner, the handbook is intended to fill a gap in the literature: There is a complete lack of language-independent guidance about how to actually capture an architecture in written form so that it can fulfill its purpose as a communication vehicle providing a unified design vision to all of the varied stakeholders of a development project.

The theme of the work is that documenting an architecture entails documenting the set of relevant views of that architecture, and then completing the picture with documentation of information that transcends any single view. The report lays out our approach and organization for the complete book, and provides full guidance for one of the most commonly used architectural views: the layer diagram. The audience for this book is the community of practicing architects, apprentice architects, and developers who are on the receiving end of architectural documentation.

14. **SUBJECT TERMS**
architectrual views, layer diagram, software architecture, software architecture documentation

15. **NUMBER OF PAGES**
29 pp.

16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT**
UNCLASSIFIED

18. **SECURITY CLASSIFICATION OF THIS PAGE**
UNCLASSIFIED

19. **SECURITY CLASSIFICATION OF ABSTRACT**
UNCLASSIFIED

20. **LIMITATION OF ABSTRACT**
UL